BELOUADAH Sakher M2 AIC

Mid-term exam 16/10/2015

# Answers:

**A-** Branch and bound is an optimization problem technique, the idea is to construct the candidate one at a time and then evaluate the partially constructed solution, we continue constructing if there's a potential value that lead to a solution, otherwise we don't develop at all! Which has a good advantage when it comes to large data problems.

**B-** The brute force method consist of constructing the whole search space, which leads to an exponential complexity while the Branch and bound method reaches solution fast by removing search space, which means if no potential values of the remaining components can lead to a solution, the remaining components are not generated at all.

Although, in the worst case scenario, breach and bound will have an exponential complexity, but this won't happen in most cases (unlike brute force).

**C-** The first thing we need to know, is how to represent a solution, in this case, a solution would be a list of ones and zeros to define which of the items is selected. The branch and bound technique can be schematized by a tree, each node is a solution with different length, and the length determines the number of items that I decided to pick or not. For example R101 means, from the root, I picked item1 and item3.

How to split the problem into sub problems? We start from the root, which is the state of picking no item, we then decide for each item, rather to add it or not to the sack, and compute each time the benefit and the weight that it adds. Therefore, at each level, we split our sub problem into even smaller one.

By adding the upper bound constraint, we substantially downgrade the search space. For each generated solution, we calculate its upper bound, and add it to a list of already generated upped bounds, we then develop the solution which have the maximum upper bound, because it's the one that can lead to the maximum benefit.

|  |
| --- |
| #  # Pseudo code:  **Inputs**:  **maxWeight** that the knapsack can carry  **Values** : an array that contains the value of each item  **Weights** : an array that contain the weight of each item  **outputs** :  **benefit**: the best benefit of the instance of problem  **weight**: the best weight reached  **structure**:  **node** : (benefit,weight,upper bound value)  **Begin**  arrange the items in descending way according to the ratio value/weight  calculate initial upper bound  initialize the root of the tree (0,0,initial upper bound)  add root to **list\_of\_best\_solutions** and define it as the best solution yet  **while** not\_empty(**list\_of\_best\_solutions**) **AND** (**best\_solution** still has items to add):  **best\_solution** = max (**list\_of\_best\_solutions.upper\_bound**)  create right child #means next item won't be added  calculate its upper bound  add it to **list\_of\_best\_solutions**  create left child if weight not exceeded !  #means next item will be added that's why we verify weight  calculate its upper bound  add it to **list\_of\_best\_solutions**  # now that the parent is developed we delete it  delete the parent from the **list\_of\_best\_solutions**  return best\_solution(**benefit**,**weight**)  **end** |

**D-**done! (basic\_branche\_and\_bound)

**E-** The idea behind calculating the upper bound is the one used in the greedy algorithm, we start by ordering the items in a descendent way according to the ratio value/weight. To calculate it, we start adding items in that specific order while the weight is not exceeded. When we reach the item Kth and we can't add it due to its weight, we calculate the value of the possible fraction that we can add from this item. Example: if the Kth item has a weight of 9, and a value of 45, and the remaining weight is 8. Then we add (8/9)\*45 to the current benefit.

We don't take in consideration the items that we did not pick when calculating the upper bound of each node, means that if the current node is R101 then we calculate the upper bound knowing that we didn't choose the second item so we can't possibly add it.

**F-**done! No external code has been used.

The following table represent a comparison of each method executed on three files with different dimensions.

10items1:

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Execution time | Optimal Benefit | Correspondent weight |
| Brute force | 6.83 ms | 308 | 249 |
| Dynamic | 4.14 ms | 308 | 249 |
| Basic B&B | 3.97 ms | 308 | 249 |
| Advanced B&B | 493 µs | 308 | 249 |
| Greedy | 11.5 µs | 298 | 201 |

20items1 : max weight = 500

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Execution time | Optimal Benefit | Correspondent weight |
| Brute force | 9.62 s | 740 | 493 |
| Dynamic | 19 ms | 740 | 475 |
| Basic B&B | >10 mnts | 740 | 475 |
| Advanced B&B | 1.83 ms | 740 | 475 |
| Greedy | 23 µs | 728 | 475 |

30items1 : max weight = 750

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Execution time | Optimal Benefit | Correspondent weight |
| Brute force | Memory Exceeded | 1026 | 747 |
| Dynamic | 44.7 ms | 1026 | 747 |
| Basic B&B | >10 mnts | 1026 | 747 |
| Advanced B&B | 5.08 ms | 1026 | 747 |
| Greedy | 31.2 µs | 1001 | 713 |

Remarque: Even though the greedy approach is the fastest, it does not lead to an optimal solution.

The first thing that we notice is that the advanced branch and bound using the subroutine approach is the fastest one when we compare it to the basic branch and bound, dynamic and brute force.

As we increase the number of items, the execution time of the brute force and basic branch and bound increases, that’s due to the number of operations that grow exponentially.

In term of the optimal benefit found for each problem, all the methods got it right and outperform the greedy approach. There is some exceptions where the greedy approach can find the optimal solution (like in the file 10items-3), but it won’t for most cases and still depends on the items and weights provided, unlike the other approaches.

We it comes to memory consumption, the dynamic programing depends on the number of items and the weight. The more items a problem have, the biggest the generated matrix get. The same applies to branch and bound approach, as its memory consumption grows as the number of items increase, and so is the number of operations.

As a conclusion, we can confirm that the choice of the method depends on the problem’s nature, and not only on its complexity as we found that other parameters might affect the computing (like the size of the population). In our case – the 1/0 knapsack problem – we found that the best methods to solve this problem are dynamic programing and advanced branch and bound using the upper bound.